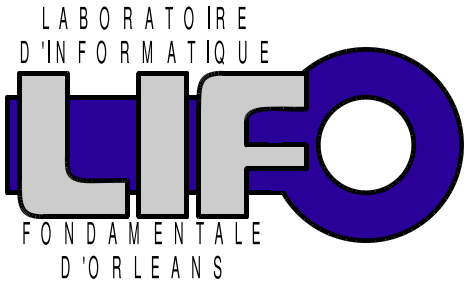

4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Minimal Extensions of Tree Languages: Application to XML Schema Evolution

Jacques Chabin, Mirian Halfeld-Ferrari,
Martin A. Musicante, Pierre Réty
LIFO, Université d'Orléans

Rapport n^o **RR-2009-06**

Minimal Extensions of Tree Languages: Application to XML Schema Evolution

Jacques Chabin
Université d'Orléans, LIFO
Orléans, France
jacques.chabin@univ-orleans.fr

Mirian Halfeld-Ferrari*
Université d'Orléans, LIFO
Orléans, France
mirian.halfeld@univ-orleans.fr

Martin A. Musicante†
Universidade Federal do Rio Grande do Norte, DIMAp
Natal, Brazil
mam@dimap.ufrn.br

Pierre Réty
Université d'Orléans, LIFO
Orléans, France
pierre.rety@univ-orleans.fr

ABSTRACT

Data shared among applications has a highly dynamic nature. The schema of this data often changes as the information grows and applications that uses an old version of the schema may need to adjust to the new one. The new schema should be an evolution of the original one, adding the capability of accepting a given new data.

In this context, we propose two automatic, conservative methods for evolving schemas (or types) for semi-structured data. The application of our method ranges from allowing the support for new data in a database to more elaborated usage, such as allowing the generalisation of different message types, for instance, in order to define the type of messages a web service should deal with.

We propose two algorithms, producing different classes of schema for XML (those classes represented, respectively by *DTD* and *XML Schema*). Each algorithm is given a regular tree grammar and returns the *least schema* of its class that contains the language of the original grammar (in the sense of language inclusion). Our algorithms are inspired by grammar inference techniques. They are proven correct.

The intended use of our method is as follows: We start with a regular tree grammar G_0 which is the union of two other grammars: (i) a given original grammar G_{orig} which can be a local tree grammar (LTG or DTD) or a single-type tree grammar (STTG or XML Schema) and (ii) the grammar $G_{newTree}$ which describes a tree that does not belong to $L(G_{orig})$ (the language generated by G_{orig}) but that represents the data for which an evolution should be triggered. Our algorithms yield the grammars (LTG or STTG according to the user's choice) generating the least languages (local tree language or single-type tree language) containing both the original language and the new tree. The paper also discusses the implementation of a prototype (proof of concept).

*Partly supported by Codex project ANR-08-DEFIS-04. Part of this work was done while the author was working at Université François Rabelais de Tours.

†Partly supported by Codex project ANR-08-DEFIS-04. This work was done while the author was visiting Université d'Orléans.

1. INTRODUCTION

Dealing with the dynamic nature and great volume of the data exchanged on the Internet requires the development of new techniques and tools. The applications that manipulate these data should be dependable [KH08]. One big step towards dependability is that the algorithms that manipulate these data are proven correct.

XML is a standard format for data exchange in many branches of science as well as in e-business/e-government. In a collaborative environment, data communication among peers should respect some type constraints. The definition of new rules for the typing of these data is not rare and the new type must be communicated among peers that already have a body of local data that complies with the old rules. Indeed, the applicability of schema evolution is very large: it can help in the automatic adaptation of those types of messages exchanged between web services; in the maintenance of data related to digital libraries or in the automatic merge of several XML data repositories, by obtaining of a common schema, build up from types originated at different sources.

In this context, the learning of new types (or schema, as it is called by the XML community) can be very helpful for the harmonious work of the applications that manipulate these data. Some algorithms for learning XML data have been proposed [GGR$^+$00, Chi01, BNV07]. In general, these algorithms use traditional learning techniques, which consist of learning the schema using sets of (positive or negative) examples. Unfortunately, the distributed and gigantic nature of the data in question makes it unpractical to use traditional techniques. Our work takes this situation into account and places ourselves in a more realistic scenario, where the data administrator needs to incorporate a new data in his application. Supposing that this new data cannot be validated wrt the current type, it would be very useful

to automatically generate a new type such that: (i) it is a conservative extension of the old one and (ii) the new data is valid wrt to it. Moreover, we are interested in obtaining the *least* schema (in the sense of type inclusion) that complies with conditions (i) and (ii) and that can be specified in current XML schema languages standards such as DTD or XMLSchema. In this way, our method allows the evolution of a given schema: usually we talk about the evolution of a DTD or an XMLSchema, but one may also provoke the evolution of a DTD towards an XMLSchema. The following example illustrates schema evolution as proposed by our method.

Example 1.1 Consider a web service that exchanges messages concerning address of people working a research laboratory. As usual we represent non terminals starting with a capital letter and terminals with a small letter. Let G be the LTG which defines the addresses of professors and students and that has D as the axiom. Besides the production rules presented below, we suppose that all the production rules $X \rightarrow x[\epsilon]$ such that $X \in \{N, M, T, A\}$ and $x \in \{name, number, telephone, address\}$ are included in the solution:

	Productions rules of G
D	$\rightarrow directory[S^*.P^*]$
S	$\rightarrow student[N.M.A?]$
P	$\rightarrow professor[N.A?.T^*]$

Now, suppose the web service should also accept the document $directory(student(name, phone))$. Clearly, this tree does not respect the type imposed by G. The administrator may require the original schema (grammar) to evolve. Grammar G' (whose production rules are presented below and whose axiom is Y) generates the new tree, but the grammar G_1 resulting from a simple union of G and G' is neither a LTG nor a STTG.

	Productions rules of G'
Y	$\rightarrow directory[B]$
B	$\rightarrow student[C.H]$
C	$\rightarrow name[\epsilon]$
H	$\rightarrow phone[\epsilon]$

Supposing a DTD is the schema language used to express the web service message type, the schema evolution should results into a new LTG grammar. Our algorithms gives the following grammar G_2 as output (by merging some production rules of G_1) with D_Y as axiom.

	Productions rules of G_2
D_Y	$\rightarrow directory[S_B \mid S_B^*.P^*]$
S_B	$\rightarrow student[N_C.T_H \mid N_C.M.A?]$
P	$\rightarrow professor[N_C.A?.T_H^*]$
N_C	$\rightarrow name[\epsilon]$
T_H	$\rightarrow phone[\epsilon]$
M	$\rightarrow number[\epsilon]$
A	$\rightarrow address\ [\epsilon]$

□

In this context, we propose algorithms to evolve schemas given as a Local Tree Grammar (*i.e.* a DTD specification) or as a Single-Type Tree Grammar (*i.e.* a XMLSchema specification). Our algorithms are capable of finding a definition for the *least* (local or single-type) tree language (set of XML documents) that contains both the XML documents described by the original type and the newly-arrived data. For instance, in Example 1.1, grammar G_2 is the least LTG that contains the language generated by G and the tree $directory(student(name, phone))$.

The algorithms proposed in this paper are able to transform a (general) regular tree grammar into a Local or Single-Type tree grammar. Our algorithms are proven correct for any regular tree grammar in reduced normal form (a relevant XML schema is in reduced form, and any regular tree grammar can be transformed into normal form).

The rest of this paper is organised as follows: in Section 2 we recall the theoretical background needed to the introduction of our method; Section 3 presents our schema evolution algorithms for LTG and STTG and Section 4 discuss the implementation of these methods. The paper finishes by considering some related work and by discussing our perspectives of work.

2. THEORETICAL BACKGROUND

This section presents theoretical notions necessary in this paper. It is a well known fact that type definitions for XML and regular tree grammars are similar notions and that some schema definition languages can be represented by using specific classes of regular tree grammars. Thus, DTD and XML Schema, correspond, respectively, to Local Tree Grammars and Single-type Tree Grammars [MLMK05].

Given a XML type T and its corresponding tree grammar G, the set of XML documents described the type T corresponds to the language (set of trees) generated by G.

Trees can be defined as *ranked* or *unranked*. In ranked trees, each node has a fixed number of children while in unranked ones each node can have a finite, not predefined, list of children. In this paper we consider a tree language as a set of *unranked* trees. Next, we recall the notions of unranked Σ-valued trees.

Let U be the set of all finite strings of non-negative integers with the empty string ϵ as the identity. In the following definition we assume that $Pos(t) \subseteq U$ is a nonempty set closed under prefixes[1], *i.e.* , if $u \preceq v$, $v \in Pos(t)$ implies $u \in Pos(t)$.

Definition 2.1 (Unranked Σ-valued tree t) A nonempty unranked Σ-valued tree t is a mapping $t : Pos(t) \rightarrow \Sigma$ where $Pos(t)$ satisfies: $j \geq 0, uj \in Pos(t), 0 \leq i \leq j \Rightarrow ui \in Pos(t)$. The set $Pos(t)$ is also called the set of *positions* of t. We write $t(v) = a$, for $v \in Pos(t)$, to indicate that the Σ-symbol associated to v is a. Define an *empty tree* t as the one having $Pos(t) = \{\epsilon\}$ and $t(\epsilon) = \lambda$ where λ is a special symbol not in Σ. □

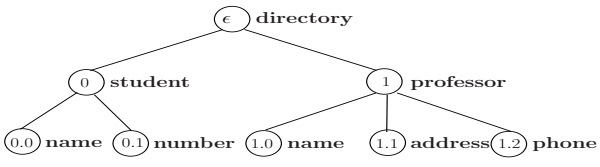

Figure 1: An XML tree.

[1]The *prefix relation* in U, denoted by $\preceq$ is defined by: $u \preceq v$ iff $uw = v$ for some $w \in U$.

Fig. 1 represents a tree whose alphabet is the set of element names appearing in an XML document. Given a tree t we denote by $t|_p$ the subtree whose root is at position $p \in Pos(t)$, *i.e.* $Pos(t|_p) = \{s \mid p.s \in Pos(t)\}$ and for each $s \in Pos(t|_p)$ we have $t|_p(s) = t(p.s)$. For instance, in Fig. 1 $t|_0 = \{(\epsilon, student), (0, name), (1, number)\}$, or equivalently, $t|_0 = student(name, number)$.

Given a tree t such that the position $p \in Pos(t)$ and a tree t', we note $t[p \leftarrow t']$ as the tree that results of substituting the subtree of t at position p by t'.

Definition 2.2 (Forest) Let L be a set of trees. $ST(L)$ denotes the set of sub-trees of elements of L, i.e. $ST(L) = \{t \mid \exists u \in L, \exists p \in Pos(u), t = u|_p\}$. A *forest* is a (possibly empty) tuple of trees. For a terminal a and a forest $w = \langle t_1, \ldots, t_n \rangle$, $a(w)$ is defined by $a(w) = a(t_1, \ldots, t_n)$. On the other hand, $w(\epsilon)$ is defined by $w(\epsilon) = \langle t_1(\epsilon), \ldots, t_n(\epsilon) \rangle$, i.e. the tuple of the top symbols of w. □

The definition of a regular grammar for unranked trees comes from its ranked version (see [CDG+02]) by allowing production rules whose right-hand side contains a regular expression [BMW01]. By convention we use small letters for terminals and capital letters for non terminals.

Definition 2.3 (Regular Tree Grammar)
A *regular tree grammar* (RTG) is a 4-tuple $G = (N, T, S, P)$, where:

- N is a finite set of *non-terminal symbols*,
- T is a finite set of *terminal symbols*,
- S is a set of *start symbols*, where $S \subseteq N$.
- P is a finite set of *production rules* of the form $X \to a\,[R]$, where $X \in N$, $a \in T$, and R is a regular expression over N (We say that, for a production rule, X is the left-hand side, $a\,R$ is the right-hand side, and R is the content model.) □

Definition 2.4 (Derivation) For a RTG $G = (N, T, S, P)$, we say that a term t build on $N \cup T$ derives (in one step) into t' iff (i) there exists a position p of t such that $t|_p = A \in N$ and a production rule $A \to a\,[R]$ in P, and (ii) $t' = t[p \leftarrow a(w)]$ where $w \in L(R)$ ($L(R)$ is the set of words of non-terminals generated by R). We note $t \to_{[p, A \to a\,[R]]} t'$. More generally, a derivation (in several steps) is a (possibly empty) sequence of one-step derivations. We note $t \to_G^* t'$. The language $L(G)$ generated by G is the set of trees containing only terminal symbols, defined by: $L(G) = \{t \mid \exists A \in S,\ A \to_G^* t\}$. □

Example 2.1
Given the tree grammar $G = (N, T, \{X\}, P)$, where $P = \{X \to f\,[A^*.B], A \to a, B \to b\}$. We have the derivation from the start symbol $X \to_{[X \to f\,[A^*.B]]} f(A, A, B) \to_G^* f(a, a, b)$. Consequently $f(a, a, b) \in L(G)$. □

To produce grammars that generate least languages, our algorithms need to start from grammars in reduced form and (as in [ML02]) in normal form.

Definition 2.5 (Reduced Form)
A *regular tree grammar* (RTG) is said to be in *reduced form* if (i) every non-terminal is reachable from a start symbol, and (ii) every non-terminal generates at least one tree containing only terminal symbols. □

Note that a relevant XML schema is already in reduced form. To check that a grammar $G = (N, T, S, P)$ is in reduced form, we compute the set of non-terminals reachable from the start symbols and check that it is equal to N (item (i)). Since emptiness is decidable for RTGs, checking (ii) amounts to check that for all $A \in N$, $G_A = (N, T, \{A\}, P)$ generates a non-empty language.

Converting an arbitrary RTG into reduced form is always possible, however it may lead to an exponential blow-up in the number of non-terminals.

Definition 2.6 (Normal Form) A regular tree grammar (RTG) is said to be in *normal form* if (i) it does not have two production rules of the form $A \to a[E_1]$ and $A \to b[E_2]$ where terminals $a \neq b$ and (ii) no two production rules have the same non-terminal in the left-hand side and the same terminal in the right-hand side. □

Notice that the definition above is closed to that in [ML02]; the difference is that we include the condition (i) to our normal form definition. The conversion of a regular tree grammar (RTG) into a normal form is done according to the following steps: (1) For every pair of rules $A \to a[E_1]$ and $A \to b[E_2]$ rewrite one of the rules by replacing the non terminal A by a new non terminal B (obtaining, for instance, $B \to b[E_2]$) and substituting all occurrences of A by $A \mid B$ in all regular expressions. If A is a start symbol, B should be added to the set of start symbols. (2) For every pair of rules $A \to a[E_1]$ and $A \to a[E_2]$ replace them by the rule $A \to a[E_1 \mid E_2]$. Note that the conversion into normal form preserves the reduced form.

Definition 2.7 (Reduced Normal Form) A regular tree grammar (RTG) is said to be in *reduced normal form* if it is both in reduced form and in normal form. □

Example 2.2 Given the tree grammar $G_0 = (N, T, S, P_0)$, where $N = \{X, A, B\}$, $T = \{f, a, c\}$, $S = \{X\}$, and $P_0 = \{X \to f\,[A.B], A \to a, B \to a, A \to c\}$. Note that G_0 is in reduced form, but it is not in normal form. By applying step (1) of the conversion method discussed above, we obtain the set $P_1 = \{X \to f\,[(A|C).B], A \to a, B \to a, C \to c\}$. Thus $G_1 = (N \cup \{C\}, T, S, P_1)$ is already in reduced normal form. □

The following three definitions come from [MLMK05].

Definition 2.8 (Competing Non-Terminals) Two different non-terminals A and B (of the same grammar G) are said *competing with each other* if

- one production rule has A in the left-hand side,
- another production rule has B in the left-hand side, and
- these two production rules share the same terminal symbol in the right-hand side. □

Definition 2.9 (Local Tree Grammar) A *local tree grammar* (LTG) is a regular tree grammar that does not have competing non-terminals.
A *local tree language* (LTL) is a language that can be generated by at least one LTG. □

Note that converting a LTG into normal form produces a LTG as well.

Definition 2.10 (Single-Type Tree Grammar) A *single-type tree grammar* (STTG) is a regular tree grammar in normal form, where (i) for each production rule, non terminals in its regular expression do not compete with each other, and (ii) starts symbols do not compete with each other. A *single-type tree language* (STTL) is a language that can be generated by at least one STTG. □

In [MLMK05] the expressive power of these classes of languages is discussed. We recall that $LTL \subset STTL \subset RTL$. Moreover, the LTL and STTL are closed under intersection but not under union; while the RTL are closed under union, intersection and difference.

Example 2.3 Let G_1, G_2 and G_3 be regular tree grammars. Each grammar G_i is defined by $G_i = (N, T, S = \{Dir\}, P_i)$. We consider that all three sets of production rules have rules $A \rightarrow a[\epsilon]$ where $a \in \{name, add, phone, number\}$ besides those specified below.

P_1	
Dir	$\rightarrow directory[Student^*.Prof^*]$
$Student$	$\rightarrow student[DirA \mid DirB])$
$Prof$	$\rightarrow professor[DirB]$
$DirA$	$\rightarrow direction[Name.Number?.Add?]$
$DirB$	$\rightarrow direction[Name.Add?.Phone^*]$

P_2	
Dir	$\rightarrow directory[Student^*.Prof^*]$
$Student$	$\rightarrow student[DirA]$
$Prof$	$\rightarrow professor[DirB]$
$DirA$	$\rightarrow direction[Name.Number.Add?]$
$DirB$	$\rightarrow direction[Name.Add?.Phone^*]$

P_3	
Dir	$\rightarrow directory[Student^*.Prof^*]$
$Student$	$\rightarrow student[Name.Number.Add?]$
$Prof$	$\rightarrow professor[Name.Add?.Phone^*]$

Notice that according to the previous definitions, grammar G_3 is a LTG, G_2 is a $STTG$ and not a LTG ($DirA$ and $DirB$ compete with each other) while G_1 is a RTG which is not a $STTG$ ($DirA$ and $DirB$ compete and belong to the same regular expression). □

3. SCHEMA EVOLUTION

Our goal is not schema extraction but to allow a given schema to evolve (in a conservative way) according to the set of new XML documents that need to be accepted by the application. This section describes our schema evolution approach by presenting the main theoretical contributions of our work.

The algorithms proposed here take as argument a general regular tree grammar in reduced normal form. They produce a LTG (respectively a STTG) whose language is the least LTL (resp. the least STTL) that contains the language described by the original tree grammar.

The intuitive idea underlying both algorithms is to identify sets of competing non-terminal symbols of the tree grammar, and fix the problem by identifying these non-terminals as the same one.

3.1 Generating LTG from RTG

We consider the problem of obtaining a local tree grammar whose language contains a given tree language. Given a regular tree grammar G_0, we are interested in the definition of the *least* local tree language that contains the language generated by G_0. The new language will be described by a local tree grammar. The algorithm described below obtains a new grammar by transforming G_0. The transformation rules are intuitively simple : every pair of competing non-terminals are transformed into one symbol. We show that this simple transformation of the original grammar yields to a local tree grammar in a finite number of steps.

Let us now consider some useful properties of local tree languages. These properties will be used to show the correctness of our grammar-transformation algorithm. The following lemma states that the type of the subtrees of a tree node is determined by the label of its node (*i.e.* the type of each node is *locally* defined). Recall that $ST(L)$ is the set of sub-trees of elements of L.

Lemma 3.1 Let L be a local tree language (LTL). Then, for each $t \in ST(L)$, each $t' \in L$ and each $p' \in Pos(t')$, we have that

$$t(\epsilon) = t'(p') \implies t'[p' \leftarrow t] \in L.$$

Proof. Let us write $a = t(\epsilon) = t'(p')$ and $t = a(w)$. L can be generated by a LTG in normal-form : $G = (N, T, S, P)$. To generate a, there is only one rule $A \rightarrow a[R] \in P$. Therefore $\exists u \in L(R)$, $u \rightarrow_G^* w$. Then $A \rightarrow_G^* a(w)$.
On the other hand, $\exists B \in S$, $B \rightarrow_G^* t'$. Consequently, $B \rightarrow_G^* t'[p' \leftarrow A] \rightarrow_G^* t'[p' \leftarrow a(w)] = t'[p' \leftarrow t]$. Thus $t'[p' \leftarrow t] \in L$. □

In the following, we also need a weaker version of the previous lemma:

Corollary 3.1 Let L be a local tree language (LTL). Then, for each $t, t' \in ST(L)$, and each $p' \in Pos(t')$, we have that

$$t(\epsilon) = t'(p') \implies t'[p' \leftarrow t] \in ST(L).$$

□

In practical terms, Corollary 3.1 give us a rule of thumb on how to "complete" a regular language in order to obtain a local tree language. For instance, let $L = \{f(a(b), c), f(a(c), b)\}$ be a regular language. According to Corollary 3.1, we know that L is not LTL and that the least local tree language L' containing L contains all trees where a has c as a child together with all trees where a has b as a child. In other words, $L' = \{f(a(b), c), f(a(c), b), f(a(c), c), f(a(b), b)\}$.

Let us now define our first algorithm for schema (DTD, Local Tree Grammar) evolution. The main intuition behind our algorithm is to merge rules having competing non terminals in their left-hand side. New non terminals are introduced in other to replace competing ones.

Definition 3.1 (Grammar Transformation) Let $G_0 = (N_0, T_0, S_0, P_0)$ be a regular tree grammar in reduced normal form. We define a new regular tree grammar $G = (N, T, S, P)$, obtained from G_0, according to the following steps:

1. Let $G_2 := G_0$, where G_2 is denoted by (N_2, T_2, S_2, P_2).
2. While there exists a pair of production rules of the form $X_1 \to a\ [R_1]$ and $X_2 \to a\ [R_2]$ in P_2 $(X_1 \neq X_2)$ do:
   (a) Let Y be a new non-terminal symbol and define a substitution $\sigma = [X_1/Y,\ X_2/Y]$.
   (b) Let $G_3 := (\sigma(N_2 \cup \{Y\}), T_2, \sigma(S_2), P_3)$, s.t. $P_3 = \sigma(P_2 \cup \{Y \to a\ [R_1|R_2]\} - \{X_1 \to a\ [R_1], X_2 \to a\ [R_2]\})$.
   (c) Let $G_2 := G_3$, where G_2 is denoted by (N_2, T_2, S_2, P_2).
3. Return G_2. □

Example 3.1 Consider the situation described in Example 1.1 where grammar G_1 is the input for the algorithm of Definition 3.1. In a first step, production rules $D \to directory[S^*.P^*]$ and $Y \to directory[B]$ are replaced by $D_Y \to directory[B \mid S^*.P^*]$. Following the same idea, rules $S \to student[N.M.A?]$ and $B \to student[C.D]$ are replaced by $S_B \to student[C.D \mid N.M.A?]$. This change implies changes on the right-hand side of other production rules such as $D_Y \to directory[S_B \mid S_B^*.P^*]$. The last steps merge rules concerning the terminals *name* and *phone*. In this way we obtain the grammar

$G_2 = (\{D_Y, S_B, P, N_C, P_D, M, A\}, \{directory, student, professor, name, phone, number, address\}, \{D_Y\}, P)$

whose set of production rules P is shown in Example 1.1. □

The next two properties are straightforward and shows that our algorithm stops, generating a local tree grammar in normal form:

Theorem 3.1 (The algorithm stops) The algorithm in Definition 3.1 always ends, producing an RTG in reduced normal form.

Proof: Straightforward, from the facts: (i) At each iteration, a pair of production rules is substituted by just one production rule. (ii) Each iteration maintains the reduced normal form of the grammar being processed. □

Theorem 3.2 The grammar produced by the algorithm of definition 3.1 is an LTG.

Proof: Straightforward, from the stop condition on the main loop. □

The following theorem is the main result of this sub-section. It shows that the language generated by the grammar obtained by algorithm in Definition 3.1 is the least possible language (in the sense of set inclusion) that is both a local tree language and contains the language generated by the original grammar G_0.

Theorem 3.3 The grammar returned by the algorithm of Definition 3.1 generates the least local tree language that contains $L(G_0)$. □

The intuition behind the proof of this theorem is as follows. Let G be the grammar produced by our algorithm and let G' be any LTG such that $L(G_0) \subseteq L(G')$,

We have to prove that $L(G_0) \subseteq L(G)$ (soundness), and that $L(G) \subseteq L(G')$ (minimality: $L(G)$ is the least LTL containing $L(G_0)$). Proving soundness is not very difficult. Minimality comes from the following steps:

- As production rules of a LTG in normal form define a bijection between the sets of terminals and non-terminals, there is only one rule in G of the form $A \to a[R]$ producing subtrees with root a. By the construction of our algorithm this rule should correspond to a rule $A_j \to a[R_j]$ in G_0 with $R = \theta(R_1) \mid \cdots \mid \theta(R_n)$ where θ is a composition of substitutions.
- Consequently, we can prove that if $a(w)$ is a subtree of $t \in L(G)$, then there is at least one tree in $L(G_0)$ with $a(w')$ as a subtree, s.t. $w'(\epsilon) = w(\epsilon)$ (i.e. forests w' and w have the same tuple of top-symbols).
- w is a forest composed by subterms of $L(G)$, and by induction hypothesis applied to each component of w (each component is a strict subtree of $a(w)$), we get that w is also a forest composed of subtrees by $L(G')$. On the other hand, since $L(G_0) \subseteq L(G')$, $a(w')$ is a subtree of $L(G')$.
- As G' is a LTG, from Corollary 3.1, we can replace each subtree of $a(w')$ –rooted by the elements of $w'(\epsilon)$– by the corresponding subtree of $a(w)$ and thus, $a(w)$ is a subtree of $L(G')$.
- Finally, as this is valid for every subtree, we have that $L(G) \subseteq L(G')$.

Formally, in order to prove Theorem 3.3, we need to prove some properties over the algorithm. Recall that $G_0 = (N_0, T_0, S_0, P_0)$ is the initial grammar considered by the algorithm, and that $G = (N, T, S, P)$ denotes the grammar computed by the algorithm.

We start by showing that our algorithm proposes a grammar which generates a language containing $L(G_0)$. The following lemma shows that the substitution defined at step 2a of the algorithm preserves the trees generated by the grammar G_0.

Lemma 3.2 Consider one step of the algorithm. If $X \to^*_{P_2} a(w)$ then $\sigma(X) \to^*_{P_3} a(w)$ where P_2 and P_3 are the set of production rules referred in Definition 3.1.

PROOF. By induction on the length of $X \to^*_{P_2} a(w)$.
We have $X \to_{P_2} a[R]$ and $\exists u \in L(R)$, $u \to^*_{P_2} w$. Two cases should be considered whether X is or not a competing non terminal.
- If X is not a competing non terminal, ie $X \notin \{X_1, X_2\}$, $\sigma(X) \to_{P_3} a\,[\sigma(R)]$ and $\sigma(u) \in L(\sigma(R))$. By induction hypothesis, $\sigma(u) \to^*_{P_3} w$, therefore $\sigma(X) \to^*_{P_3} a(w)$.
- Otherwise, suppose X is a competing non terminal, ie $X = X_1$ (the case $X = X_2$ is similar). Now $\sigma(X) = Y = \sigma(Y) \to_{P_3} a\,[\sigma(R_1|R_2)]$, and $\sigma(u) \in L(\sigma(R_1))$. By induction hypothesis, $\sigma(u) \to^*_{P_3} w$, therefore $\sigma(X) \to^*_{P_3} a(w)$. □

Example 3.2 Let the RTG G_0 be the input of the algorithm of Definition 3.1 and G be the resulting LTG:

G_0:	G:
$S \to a[A.A]$	$Y \to a[Y.Y \mid B]$
$A \to a[B]$	$B \to b[\epsilon]$
$B \to b[\epsilon]$	

Notice that $L(G_0)$ contains just the tree $t = a(a(b), a(b))$ and that $t \in L(G)$. □

We can now begin to show the relationship between the original language, as described by the grammar G_0 and the language generated by the grammar obtained by the algorithm of Definition 3.1. Let θ be the composition of all the substitutions defined by the algorithm at step 2a.

Corollary 3.2

- $L(G_0) \subseteq L(G)$
- If $X \rightarrow^*_{G_0} t$ and $\theta(X) \rightarrow^*_G t'$, then $t(\epsilon) = t'(\epsilon)$ (i.e. t and t' have the same top symbol).

PROOF. The first item is an immediate consequence of the previous lemma.
Since G_0 is in normal form, X occurs once as the left-hand side of a production rule in G_0. The same holds for $\theta(X)$ in G. Therefore X produces in G_0 only one top symbol, and $\theta(X)$ produces in G only one top symbol. Because of Lemma 3.2, these top symbols are equal. □

Notice that from the grammars of Example 3.2, we can derive $A \rightarrow^*_{G_0} a(b)$ and $X \rightarrow^*_G a(b)$ (recall that $X = \theta(A)$).

The next lemma states that the resulting grammar G does not introduce any new terminal symbol at the root of the generated trees:

Lemma 3.3 $\forall t \in L(G), \exists t' \in L(G_0), t'(\epsilon) = t(\epsilon)$.

PROOF. Let $t = a(w) \in L(G)$. The sets of initial non-terminals satisfy $S = \theta(S_0)$. Therefore there exists $A \in S_0$ s.t. $\theta A \rightarrow_G a(u)$ with $u \in N^*$ and $u \rightarrow^*_G w$.
Since G_0 is in reduced form, there exist $A \rightarrow a'[R] \in P_0$, $v \in L(R)$, and a forest w' s.t. $v \rightarrow^*_{G_0} w'$. Therefore $A \rightarrow^*_{G_0} a'(w')$.
From Corollary 3.2, $a' = a$, then there exists $t' = a(w') \in L(G_0)$ and $t'(\epsilon) = a = t(\epsilon)$. □

Next, we show that for every sub-tree generated by G, its root appears in at least one subtree of the language generated by G_0 (recall that $w'(\epsilon) = w(\epsilon)$ means that forests w' and w have the same tuple of top-symbols):

Lemma 3.4 If $t \in ST(L(G))$, such that $t = a(w)$, then, $\exists t' \in ST(L(G_0))$, $t' = a(w') \wedge w'(\epsilon) = w(\epsilon)$.

PROOF. Let $t \in ST(L(G))$ s.t. $t = a(w)$. There exists $A \rightarrow a[R] \in P$ s.t. $A \rightarrow a(u)$, $u \in L(R)$, $u \rightarrow^*_G w$. By construction, $R = \theta(R_1)|\dots|\theta(R_n)$, and $\forall i$, $\exists A_i \in N_0$, $A_i \rightarrow a[R_i] \in P_0 \wedge \theta(A_i) = A$. Therefore there exists j s.t. $u \in L(\theta(R_j))$. Consider $A_j \rightarrow a[R_j] \in P_0$. There exists $u' \in L(R_j)$ s.t. $\theta(u') = u$.
Since G_0 is in reduced form, there exists a forest w' s.t. $u' \rightarrow^*_{G_0} w'$. Consequently $A_j \rightarrow_{G_0} a(u') \rightarrow^*_{G_0} a(w')$. Let $t' = a(w')$. Since G_0 is in reduced form, the rule $A_j \rightarrow a[R_j]$ is reachable in G_0, then $t' \in ST(L(G_0))$. From Corollary 3.2, since $\theta(u') = u$, we have $w(\epsilon) = w'(\epsilon)$. □

As an illustration of Lemma 3.4, we observe that from the grammars of Example 3.2, given the tree $a(a(b), a(a(b), a(b)))$, for its sub-tree $t = a(a(b), a(b)) \in ST(L(G))$, we have $t' = t \in ST(L(G_0))$.

Now, we can prove that the algorithm just adds what is necessary to get a LTL (and not more), in other words, that $L(G)$ is the least local tree language that contains $L(G_0)$. This is done in two stages: first for subtrees, then for trees.

Lemma 3.5 Let L' be a LTL such that $L(G_0) \subseteq L'$. Then $ST(L(G)) \subseteq ST(L')$.

PROOF. By structural induction on the trees in $ST(L(G))$. Let $t = a(w) \in ST(L(G))$. From Lemma 3.4, there exists $t' \in ST(L(G_0))$ s.t. $t' = a(w') \wedge w'(\epsilon) = w(\epsilon)$. Since $L(G_0) \subseteq L'$, we have $ST(L(G_0)) \subseteq ST(L')$, then $t' \in ST(L')$.
If w is an empty forest (i.e. the empty tuple), w' is also empty, therefore $t = t' \in ST(L')$.
Otherwise, let us write $w = (a_1(w_1), \dots, a_n(w_n))$ and $w' = (a_1(w'_1), \dots, a_n(w'_n))$ (since $w(\epsilon) = w'(\epsilon)$, w and w' have the same top symbols). Since $t = a(w) \in ST(L(G))$, for each $j \in \{1, \dots, n\}$, $a_j(w_j) \in ST(L(G))$, then by induction hypothesis $a_j(w_j) \in ST(L')$.
L' is a LTL, and for each j we have : $a_j(w_j) \in ST(L')$, $t' \in ST(L')$, $(a_j(w_j))(\epsilon) = a_j = t'(j)$. By applying Corollary 3.1 n times, we get $t'[1 \leftarrow a_1(w_1)] \dots [n \leftarrow a_n(w_n)] = t \in ST(L')$. □

Theorem 3.4 Let L' be a LTL such that $L' \supseteq L(G_0)$. Then $L(G) \subseteq L'$.

PROOF. Let $t \in L(G)$. Then $t \in ST(L(G))$. From Lemma 3.5, $t \in ST(L')$. On the other hand, from Lemma 3.3, there exists $t' \in L(G_0)$ s.t. $t'(\epsilon) = t(\epsilon)$. Then $t' \in L'$. From Lemma 3.1, $t'[\epsilon \leftarrow t] = t \in L'$. □

This result ensures that the grammar G of Example 3.2 generates the least LTL that contains $L(G_0)$.

3.2 Generating STTG from RTG

Let us now consider the problem of obtaining a single-type tree grammar whose language contains a given tree language. Thus, given a regular tree grammar G_0, we are interested in the definition of the *least* single-type tree language that contains the language generated by G_0. The new language will be described by a single-type grammar. The algorithm described below obtains a new grammar by transforming G_0. Roughly speaking, for each production rule $A \rightarrow a\,[R]$ in G_0, an equivalence relation is defined on the non-terminals of R, so that all competing non-terminals of R are in the same equivalence class. These equivalence classes are the non-terminals of the new grammar.

Let $G_0 = (N_0, T_0, S_0, P_0)$ be a RTG in reduced normal form.

Definition 3.2 (Grouping competing non-terminals) Let $\parallel$ be the relation on N_0 defined by: for all $A, B \in N_0$, $A \parallel B$ iff $A = B$ or A and B are competing in P_0.

For any $\chi \in \mathcal{P}(N_0)$, let $\parallel_\chi$ be the restriction of $\parallel$ to the set χ ($\parallel_\chi$ is defined only for elements of χ).

Lemma 3.6 Since G_0 is in normal form, $\parallel_\chi$ is an equivalence relation for any $\chi \in \mathcal{P}(N_0)$.

Proof: Reflexivity and symmetry are obvious. Let us prove transitivity. Let $A, B, C \in \chi$ and suppose $A \parallel_\chi B$ and $B \parallel_\chi C$. If $A = B$ or $B = C$, we get automatically $A \parallel_\chi C$. Otherwise A and B are competing, and B and C are competing. Therefore there are production rules in P_0 :

$$A \rightarrow a[R_1],\ B \rightarrow a[R_2],\ B \rightarrow b[R_3],\ C \rightarrow b[R_4]$$

Since G_0 is in normal form, $a = b$, then A and C are competing, thus $A \parallel_\chi C$. □

Notations

- For any regular expression R, $N(R)$ denotes the set of non-terminals occurring in R.
- For any $\chi \in \mathcal{P}(N_0)$ and any $A \in \chi$, $\hat{A}^\chi$ denotes the equivalence class of A w.r.t. relation $\|_\chi$. In other words, $\hat{A}^\chi$ contains A and the non-terminals of χ that are competing with A in P_0.
- $\sigma_{N(R)}$ is the substitution defined over $N(R)$ by $\forall A \in N(R)$, $\sigma_{N(R)}(A) = \hat{A}^{N(R)}$. By extension, $\sigma_{N(R)}(R)$ is the regular expression obtained from R by replacing each non-terminal A in R by $\sigma_{N(R)}(A)$.

To make presentation easier, the algorithm below is not optimized. An optimized version is given in Section 4.

Definition 3.3 (RTG into STTG Transformation) Let $G_0 = (N_0, T_0, S_0, P_0)$ be a regular tree grammar in reduced normal form. We define a new regular tree grammar $G = (N, T, S, P)$, obtained from G_0, according to the following steps:

1. Let $G = (\mathcal{P}(N_0), T_0, S, P)$ where:

   $S = \{\hat{A}^{S_0} \mid A \in S_0\}$,

   $P = \{\, \{A_1, \ldots, A_n\} \to a\,[\sigma_{N(R)}(R)] \mid$
   $A_1 \to a[R_1], \ldots, A_n \to a[R_n] \in P_0,\ R = (R_1 | \cdots | R_n)\}$,
   where $\{A_1, \ldots, A_n\}$ indicates all non-empty sets containing competing non-terminals (not only the maximal ones).

2. Remove unreachable non-terminals and unreachable rules in G, then return it.

□

Our generation of STTG from RTG is based on grouping competing non-terminals into equivalence classes. In the new grammar each non-terminal is a set of non-terminals of N_0. When competing non-terminals which appear in the same regular expression R in G_0 are identified, the set that contains all of them forms a new non-terminal symbol. The production rule having this new symbol as its left-hand side is obtained from those rules containing the competing symbols in G_0.

Notice that our algorithm first considers (step 1 in Definition 3.3) *all* sets of competing non-terminals on their left-hand side (and not only maximal sets) to build the production rules of G. Thus, at step 1, G may create unreachable rules (from the start symbols), which are then removed at step 2.

Example 3.3 Consider a non-STTG grammar G_0 having the following set P_0 of productions rules (*Image* is the start symbol):
$Image \to image[Frame1 \mid Frame2 \mid Background.Foreground]$
$Frame1 \to frame[Frame1.Frame1 \mid \epsilon]$
$Frame2 \to frame[Frame2.Frame2.Frame2 \mid \epsilon]$
$Background \to back[Frame1]$
$Foreground \to fore[Frame2]$

Grammar G_0 defines different ways of decomposing an image: recursively into two or three frames or by describing the background and the foreground separately. Moreover, the background (resp. the foreground) is described by binary decompositions (resp. ternary decompositions). In other words, the language of G_0 contains the union of the trees: *image(bin(frame))*; *image(ter(frame))* and *image (back (bin (frame)), fore (ter (frame)))* where *bin* (resp. *ter*) denotes the set of all binary (resp. ternary) trees that contains only the symbol *frame*.
The algorithm returns G, which contains the rules below (the start symbol is $\{Image\}$):

$\{Image\} \to image[\{Frame1, Frame2\} \mid \{Frame1, Frame2\}$
$\quad \mid \{Background\}.\{Foreground\}]$
$\{Background\} \to back[\{Frame1\}]$
$\{Foreground\} \to fore[\{Frame2\}]$
$\{Frame1, Frame2\} \to frame[\epsilon$
$\quad \mid \{Frame1, Frame2\}.\{Frame1, Frame2\}$
$\quad \mid \{Frame1, Frame2\}.\{Frame1, Frame2\}.\{Frame1, Frame2\}]$
$\{Frame1\} \to frame[\{Frame1\}.\{Frame1\} \mid \epsilon]$
$\{Frame2\} \to frame[\{Frame2\}.\{Frame2\}.\{Frame2\} \mid \epsilon]$

Note that some regular expressions could be simplified. G is a STTG that generates the union of *image(tree(frame))* and *image (back (bin (frame)), fore (ter (frame)))* where *tree* denotes the set of all trees that contain only the symbol *frame* and such that each node has 0 or 2 or 3 children. Let $L_G(X)$ denote the language obtained by deriving in G the non-terminal X. Actually, $L_G(\{Frame1, Frame2\})$ is the least STTL that contains $L_{G_0}(Frame1) \cup L_{G_0}(Frame2)$. □

Theorem 3.5 The grammar returned by the algorithm of Definition 3.3 generates the least STTL that contains $L(G_0)$.

The rest of this sub-section consists in proving the previous theorem. The notations are those of Definition 3.3. The proof somehow looks like the proof concerning the transformation of a RTG into a LTG (Sub-section 3.1). However it is more complicated since in a STTL (and unlike what happens in a LTL), the confusion between $t|_p = a(w)$ and $t'|_{p'} = a(w')$ should be done only if position p in t has been generated by the same production rule as position p' in t', i.e. the symbols occurring in t and t' along the paths going from root to p (resp. p' in t') are the same. This is why we introduce notation $path(t, p)$ to denote these symbols (Definition 3.4).

Lemma 3.7 Let $\chi \in \mathcal{P}(N_0)$ and $A, B \in \chi$. Then $\hat{A}^\chi$ and $\hat{B}^\chi$ are not competing in P.

Proof. By contradiction. Suppose $\hat{A}^\chi$ and $\hat{B}^\chi$ are competing in P. Then there exist $\hat{A}^\chi \to a[R_1] \in P$ and $\hat{B}^\chi \to a[R_2] \in P$.
From the construction of P, there exist $C \in \hat{A}^\chi$ (then $C \|_\chi A$) and $C \to a[R'_1] \in P_0$, as well as $D \in \hat{B}^\chi$ (then $D \|_\chi B$) and $D \to a[R'_2] \in P_0$.
Therefore $C \|_\chi D$ and by transitivity $A \|_\chi B$, then $\hat{A}^\chi = \hat{B}^\chi$, which is impossible since by definition, competing non-terminals are not equal. □

Example 3.4 Given the grammar of Example 3.3, let $\chi = \{Frame1, Frame2, Background\}$. The equivalence classes induced by $\|_\chi$ are $\widehat{Frame1}^\chi = \widehat{Frame2}^\chi = \{Frame1, Frame2\}$;

$\widehat{Background}^{\chi} = \{Background\}$; which are non-competing non-terminals in P.

Lemma 3.8 G is a STTG.

PROOF. We first prove that there are no regular expression in P containing competing non-terminals.
Let $\hat{A}^{S_0}, \hat{B}^{S_0} \in S$. Then $A, B \in S_0$, and from previous lemma $\hat{A}^{S_0}$ and $\hat{B}^{S_0}$ are not competing in P. For any regular expression R, let $\hat{A}^{N(R)}, \hat{B}^{N(R)} \in N(\sigma_{N(R)}(R))$. Then $A, B \in N(R)$, and from previous lemma $\hat{A}^{N(R)}$ and $\hat{B}^{N(R)}$ are not competing in P.

It rest us to prove that G is in normal form: There is at most one rule in P whose left-hand-side is $\{A_1, \ldots, A_n\}$, since for each A_i there is at most one rule in P_0 whose left-hand-side is A_i (because G_0 is in normal form). Therefore G is in normal form. □

The next lemma stablishes the basis for proving that the language generated by G contains the language generated by G_0. It considers the derivation process over G_0 at any step (supposing that this step is represented by a derivation tree t) and proves that, in this case, at the same derivation step over G, we can obtain a tree t' having all the following properties: (i) the set of positions is the same for both trees ($Pos(t) = Pos(t')$); (ii) positions associated to terminal are identical in both trees; (iii) if position p is associated to a non-terminal A in t then position $p \in Pos(t')$ is associated to the equivalence class $\hat{A}^{\chi}$ for some $\chi \in \mathcal{P}(N_0)$ such that $A \in \chi$.

Lemma 3.9 Let $Y \in S_0$. If G_0 derives:
$t_0 = Y \to \cdots \to t_{n-1} \to_{[p_n,\, A_n \to a_n[R_n]]} t_n$ then G can derive:
$t'_0 = \hat{Y}^{S_0} \to \cdots \to t'_{n-1} \to_{[p_n,\, \hat{A_n}^{\chi_n} \to a_n[\sigma_{N(R_n|\cdots)}(R_n|\cdots)]]} t'_n$
s.t. $\forall i \in \{0, \ldots, n\}, Pos(t'_i) = Pos(t_i) \wedge$
$\forall p \in Pos(t_i)$: $(t_i(p) \in T_0 \implies t'_i(p) = t_i(p)) \wedge$
$\quad (t_i(p) = A \in N_0 \implies \exists \chi \in \mathcal{P}(N_0),\, A \in \chi \wedge t'_i(p) = \hat{A}^{\chi})$

PROOF. The proof is by induction in the length of the derivation process.
- $n = 0$. The property holds because $t_0(\epsilon) = Y$ and $t'_0(\epsilon) = \hat{Y}^{\chi}$ with $\chi = S_0$ and $Y \in \chi$.
- Induction step. Assume the property for $n - 1 \in \mathbb{N}$. By hypothesis $t_{n-1} \to_{[p_n,\, A_n \to a_n[R_n]]} t_n$, then $t_{n-1}(p_n) = A_n \in N_0$.
By induction hypothesis, $t'_{n-1}(p_n) = \hat{A_n}^{\chi_n}$ for some $\chi_n \in \mathcal{P}(N_0)$, and $A_n \in \chi_n$.
By construction of P, $\hat{A_n}^{\chi_n} \to a_n[\sigma_{N(R_n|\cdots)}(R_n|\cdots)] \in P$.
Therefore $t'_{n-1} \to_{[p_n,\, \hat{A_n}^{\chi_n} \to a_n[\sigma_{N(R_n|\cdots)}(R_n|\cdots)]]} t'_n =$
$t'_{n-1}[p_n \leftarrow a_n[\sigma_{N(R_n|\cdots)}(w)]]$ whereas $t_n = t_{n-1}[p_n \leftarrow a_n(w)]$ and $w \in L(R_n)$. Consequently
$t'_n(p_n) = a_n = t_n(p_n)$ and $\forall i \in \mathbb{N}$, $t_n(p_n.i) = B \in N(R_n) \subseteq N(R_n|\cdots) \wedge t'_n(p_n.i) = \hat{B}^{N(R_n|\cdots)}$. □

The next example illustrates Lemma 3.9.

Example 3.5 Given the grammar of Example 3.3, consider trees t, t' and t'' in Figure 2 obtained after three steps in the derivation process: t is a derivation tree for G_0 while t' and t'' are for G. Tree t' is the one that corresponds to t according to Lemma 3.9. Notice that t'' is a tree that can also be derived from G, but it is not in $L(G_0)$ (indeed, since $Pos(t) \neq Pos(t'')$, tree t'' does not have the properties required in Lemma 3.9). □

The following corollary proves that the language of the new grammar G, proposed by the algorithm of Definition 3.3, contains the original language of G_0.

Corollary 3.3 $L(G_0) \subseteq L(G)$. □

In the rest of this section we work on proving that $L(G)$ is the least STTL that contains $L(G_0)$. To prove this property, we first need to prove some properties over STTLs.

We start by considering paths in a tree. We are interested by paths starting on the root and achieving a given position p in a tree t. Paths are defined as a sequence of labels. For example, $path(a(b, c(d)),\, 1) = a.c$.

Definition 3.4 (Path in a tree t to a position p) Let t be a tree and $p \in Pos(t)$. We define $path(t, p)$ as being the word of symbols occurring in t along the branch going from the root to position p. Formally, $path(t, p)$ is recursively defined by :

- $path(t, \epsilon) = t(\epsilon)$
- $path(t, p.i) = path(t, p).t(p.i)$ where $i \in \mathbb{N}$. □

Given a STTG G, let us consider the derivation process of two trees t and t' belonging to $L(G)$. The following lemma proves that positions (p in t and p' in t') having identical paths are derived by using the same rules. A consequence of this lemma (when $t' = t$ and $p' = p$) is the well known result about the unicity in the way of deriving a given tree with a STTG [ML02].

Lemma 3.10 Let G' be a STTG, let $t, t' \in L(G')$.
Let $X \to^*_{[p_i, rule_{p_i}]} t$ be a derivation of t and $X' \to^*_{[p'_i, rule'_{p'_i}]} t'$ be a derivation of t' by G' (X, X' are start symbols). Then
$\forall p \in Pos(t)$, $\forall p' \in Pos(t')$,
$(path(t, p) = path(t', p') \implies rule_p = rule'_{p'})$

PROOF. Suppose $path(t, p) = path(t', p')$. Then we have $length(p) = length(p')$. The proof is by induction on $length(p)$.
- If $length(p) = 0$, then $p = p' = \epsilon$, and $t(\epsilon) = t'(\epsilon) = a$.
Therefore $rule_\epsilon = (X \to a[R])$ and $rule'_\epsilon = (X' \to a[R'])$.
If $X \neq X'$ then two start symbols are competing, which is impossible since G' is a STTG.
If $X = X'$ and $R \neq R'$ then G' is not in normal form, which contradicts the fact that G' is a STTG.
Therefore $rule_\epsilon = rule'_\epsilon$, then $rule_p = rule'_{p'}$.
- Induction step. Suppose $p = q.k$ and $p' = q'.k'$ ($k, k' \in \mathbb{N}$), and $path(t, p) = path(t', p')$.
Then $path(t, q) = path(t', q')$. By induction hypothesis, $rule_q = rule'_{q'} = (X \to a[R])$. There exits $w, w' \in L(R)$ s.t. $w(k) = A$, $w'(k') = A'$ and $rule_p = (A \to b[R_1])$, $rule'_{p'} = (A' \to b[R'_1])$ where $b = t(p) = t'(p')$.
If $A \neq A'$ then $A \in N(R)$ and $A' \in N(R)$ are competing, which is impossible since G' is a STTG.
If $A = A'$ and $R_1 \neq R'_1$, then G' is not in normal form, which contradicts the fact that G' is a STTG.
Consequently $rule_p = rule'_{p'}$. □

In a STTL, it is possible to permute sub-trees that have the same paths.

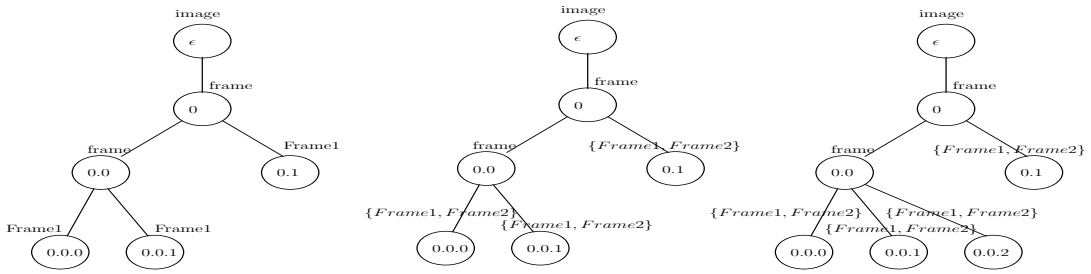

Figure 2: Derivation trees t, t' and t'' .

Lemma 3.11 Let G' be a STTG. $\forall t, t' \in L(G')$, $\forall p \in Pos(t)$, $\forall p' \in Pos(t')$, $(path(t,p) = path(t',p') \implies t'[p' \leftarrow t|_p] \in L(G'))$

Proof. Let us write $t|_p = a(f)$ and $t'|_{p'} = a(f')$ (f, f' are forests).
From lemma 3.10, $t(p)$ and $t'(p')$ have been generated by the same rule, say $X \to a[R]$. Then there exist $w, w' \in L(R)$ s.t. $w \to^*_{G'} f$ and $w' \to^*_{G'} f'$. We have (A is a start symbol)

$$A \to^*_{G'} t'[p' \leftarrow a(w')] \to^*_{G'} t'[p' \leftarrow a(f')] = t'$$

Therefore $A \to^*_{G'} t'[p' \leftarrow a(w)] \to^*_{G'} t'[p' \leftarrow a(f)] = t'[p' \leftarrow t|_p] \in L(G')$. □

Example 3.6 Let G be the grammar of Example 3.3. Consider a tree t as shown in Figure 3. The permutation of sub-trees $t|_{0.0}$ and $t|_{0.1}$ gives us a new tree t'. Both t and t' are in $L(G)$. □

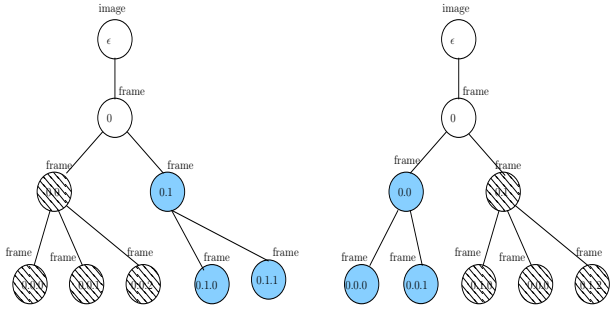

Figure 3: Trees t and t' with permuted sub-trees.

Definition 3.5 (Branch derivation) Let G' be a RTG. A *branch-derivation* is a tuple of production rules[2] of G': $\langle A^1 \to a^1[R^1], \ldots, A^n \to a^n[R^n] \rangle$ s.t. $\forall i \in \{2, \ldots, n\}, A^i \in N(R^{i-1})$. □

Notice that if A^1 is a start symbol, the branch-derivation represents the derivation of a branch of a tree[3]. This branch contains the terminals $a^1, \ldots, a^n$ (the path to the node having a^n as label).

[2]Indices are written as super-scripts for coherence with the notations used in lemma 3.12.
[3]This tree may contain non-terminals.

Now, let us prove properties over the grammar G built by Definition 3.3.

Lemma 3.12 Consider a branch-derivation in G:
$\langle \{A^1_1, \ldots, A^1_{n_1}\} \to a^1 \sigma_{N(R^1_1|\cdots|R^1_{n_1})}[R^1_1|\cdots|R^1_{n_1}], \ldots,$
$\{A^k_1, \ldots, A^k_{n_k}\} \to a^k \sigma_{N(R^k_1|\cdots|R^k_{n_k})}[R^k_1|\cdots|R^k_{n_k}] \rangle$ and let $i_k \in \{1, \ldots, n_k\}$.
Then there exists a branch-derivation in G_0:
$(A^1_{i_1} \to a^1[R^1_{i_1}], \ldots, A^k_{i_k} \to a^k[R^k_{i_k}])$.

Proof. By induction on k.
- $k = 1$. There is one step. From Definition 3.3, $A^k_{i_k} \to a^k[R^k_{i_k}] \in P_0$.
- Induction step. By induction hypothesis applied on the last $k-1$ steps, there exists a branch-derivation in G_0: $(A^2_{i_2} \to a^2[R^2_{i_2}], \ldots, A^k_{i_k} \to a^k[R^k_{i_k}])$.
Moreover $\{A^2_1, \ldots, A^2_{n_2}\} \in N(\sigma_{N(R^1_1|\cdots|R^1_{n_1})}(R^1_1|\cdots|R^1_{n_1}))$.
Then there exists $i_1 \in \{1, \ldots, n_1\}$ s.t. $A^2_{i_2} \in N(R^1_{i_1})$. And from Definition 3.3, $A^1_{i_1} \to a^1[R^1_{i_1}] \in P_0$. □

The following example illustrates Lemma 3.12 and its proof.

Example 3.7 Let G be the grammar of Example 3.3 and t the tree of Figure 3. The branch-derivation corresponding to the node 0.0.0 contains the first and the fourth rules of G presented in Example 3.3 (notice that the fourth rule appears three times). Figure 4 illustrates this branch-derivation on a derivation tree. For instance, the first rule in G is

$$\{Image\} \to image[\{Frame1, Frame2\} \mid \{Frame1, Frame2\} \mid \{Background\}.\{Foreground\}] \quad \text{(R1)}$$

and G_0 has the production rule

$$Image \to image[Frame1 \mid Frame2 \mid Background.Foreground]$$

Then, the branch-derivation gives us the fourth rule in G, namely:

$$\{Frame1, Frame2\} \to frame[\epsilon \mid \{Frame1, Frame2\}.\{Frame1, Frame2\} \mid \{Frame1, Frame2\}.\{Frame1, Frame2\}.\{Frame1, Frame2\}].$$

Notice that the left-hand side $\{Frame1, Frame2\}$ is a non terminal in the right-hand side of (R1). Now, consider each non terminal of G_0 forming the non terminal $\{Frame1, Frame2\}$ in G. Clearly, $Frame1$ is on the right-hand side of the second rule in P_0 while $Frame2$ is on the right-hand side of the third rule in P_0 (as shown in Example 3.3). We can observe the same situation for all the rules in the branch-derivation. Thus, as proved in Lemma 3.12, the branch-derivation in G_0 that corresponds to the one considered in this example is:

$\langle\ Image \to image[Frame1 \mid Frame2 \mid Background.Foreground]$
$Frame2 \to frame[Frame2.Frame2.Frame2 \mid \epsilon]$
$Frame2 \to frame[Frame2.Frame2.Frame2 \mid \epsilon]$
$Frame2 \to frame[Frame2.Frame2.Frame2 \mid \epsilon]\ \rangle$ □

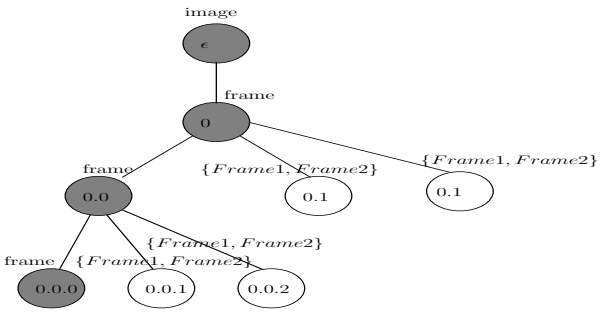

Figure 4: Derivation tree in G. Grey nodes illustrate a branch-derivation.

The following lemma somehow expresses what the algorithm of Definition 3.3 does. Given a forest $w = (t_1, \ldots, t_n)$, recall that $w(\epsilon) = \langle t_1(\epsilon), \ldots, t_n(\epsilon)\rangle$, i.e. $w(\epsilon)$ is the tuple of the top symbols of w.

Lemma 3.13 $\forall t \in L(G), \forall p \in Pos(t)$,
$t|_p = a(w) \implies \exists t' \in L(G_0), \exists p' \in pos(t'), t'|_{p'} = a(w') \wedge w'(\epsilon) = w(\epsilon) \wedge path(t', p') = path(t, p)$. □

Proof. There exists a branch-derivation in G that derives the position p of t
$(\{A^1_1, \ldots, A^1_{n_1}\} \to a^1\sigma_{N(R^1_1|\cdots|R^1_{n_1})}[R^1_1|\cdots|R^1_{n_1}], \ldots,$
$\{A^k_1, \ldots, A^k_{n_k}\} \to a^k\sigma_{N(R^k_1|\cdots|R^k_{n_k})}[R^k_1|\cdots|R^k_{n_k}])$
and $u \in L(\sigma_{N(R^k_1|\cdots|R^k_{n_k})}(R^k_1|\cdots|R^k_{n_k}))$ s.t. $u \to^*_G w$.
Then there exists i_k s.t. $u \in L(\sigma_{N(R^k_1|\cdots|R^k_{n_k})}(R^k_{i_k}))$.
Then there exists $v \in L(R^k_{i_k})$ s.t. $u = \sigma_{N(R^k_1|\cdots|R^k_{n_k})}(v)$.
Note that $\forall Y \in N_0, \forall \chi \in \mathcal{P}(N_0)$, Y and $\hat{Y}^\chi$ generate the same top-symbol. So u and v generate the same top-symbols. Since G_0 is in reduced form, there exists w' s.t. $v \to^*_{G_0} w'$, and then $w'(\epsilon) = w(\epsilon)$.
From Lemma 3.12, there exists a branch-derivation in G_0 : $(A^1_{i_1} \to a^1[R^1_{i_1}], \ldots, A^k_{i_k} \to a^k[R^k_{i_k}])$.
Since G_0 is in reduced form, there exists $t' \in L_{G_0}(A^1_{i_1})$ (i.e. t' is a tree derived from $A^1_{i_1}$ by rules in P_0, and t' contains only terminals), and there exists $p' \in Pos(t')$ s.t. this branch-derivation derives in G_0 the position p' of t'.
Since $v \in L(R^k_{i_k})$ and $v \to^*_{G_0} w'$, one can even choose t' s.t. $t'|_{p'} = a^k(w')$. Since $a^k = a$, we have $t'|_{p'} = a(w')$.
On the other hand, $path(t', p') = a^1 \ldots a^k = path(t, p)$.
Finally, since $t \in L(G)$, $\{A^1_1, \ldots, A^1_{n_1}\} \in S$. Since $A^1_{i_1} \in \{A^1_1, \ldots, A^1_{n_1}\}$, from Definition 3.3 we have $A^1_{i_1} \in S_0$. Therefore $t' \in L(G_0)$. □

Example 3.8 Let G be the grammar of Example 3.3 and t the tree of Figure 3. Let $p = 0$.
Using the notations of Lemma 3.13, $t|_0 = frame(w)$ where $w = \langle frame(frame, frame, frame),\ frame(frame, frame)\rangle$.
$t \notin L(G_0)$, however
$t' = image(frame(frame(frame, frame),\ frame)) \in L(G_0)$
and (with $p' = p = 0$) $t'|_{p'} = frame(w')$ where $w' = \langle frame(frame, frame),\ frame\rangle$. Thus $w'(\epsilon) = w(\epsilon)$.
Note that some others $t' \in L(G_0)$ suit as well.

We end this section by proving that the grammar obtained by our algorithm generates the least STTL which contains $L(G_0)$.

Lemma 3.14 Let L' be a STTL s.t. $L(G_0) \subseteq L'$. Let $t \in L(G)$. Then, $\forall p \in Pos(t), \exists t' \in L', \exists p' \in pos(t')$, $(t'|_{p'} = t|_p \wedge path(t', p') = path(t, p))$. □

Proof. We define the relation $\sqsupset$ over $Pos(t)$ by $p \sqsupset q \iff \exists i \in \mathbb{N}, p.i = q$. Since $Pos(t)$ is finite, $\sqsupset$ is noetherian. The proof is by noetherian induction on $\sqsupset$.
Let $p \in pos(t)$. Let us write $t|_p = a(w)$.
From Lemma 3.13, we know that:
$\exists t' \in L(G_0), \exists p' \in pos(t'), t'|_{p'} = a(w') \wedge w'(\epsilon) = w(\epsilon)$
$\wedge path(t', p') = path(t, p)$. Thus, $t|_p = a(a_1(w_1), \ldots, a_n(w_n))$ and $t'|_{p'} = a(a_1(w'_1), \ldots, a_n(w'_n))$.

Now let $p \sqsupset p.1$. By induction hypothesis:
$\exists t'_1 \in L', \exists p'_1 \in pos(t'_1), t'_1|_{p'_1} = t|_{p.1} = a_1(w_1) \wedge path(t'_1, p'_1) = path(t, p.1)$. Notice that $t'_1 \in L', t' \in L(G_0) \subseteq L'$, and L' is a STTL. Moreover $path(t'_1, p'_1) = path(t, p.1) = path(t, p).a_1 = path(t', p').a_1 = path(t', p'.1)$.
As $path(t'_1, p'_1) = path(t', p'.1)$, from Lemma 3.11 applied on t'_1 and t', we get $t'[p'.1 \leftarrow t'_1|_{p'_1}] \in L'$.
However $(t'[p'.1 \leftarrow t'_1|_{p'_1}])|_{p'} = a(a_1(w_1), a_2(w'_2), \ldots, a_n(w'_n))$ and $path(t'[p'.1 \leftarrow t'_1|_{p'_1}], p') = path(t', p') = path(t, p)$.
By applying the same reasoning for positions $p.2, \ldots, p.n$, we get a tree $t'' \in L'$ s.t. $t''|_{p'} = t|_p$ and $path(t'', p') = path(t, p)$. □

Corollary 3.4 (when $p = \epsilon$, and then $p' = \epsilon$)
Let L' be a STTL s.t. $L' \supseteq L(G_0)$. Then $L(G) \subseteq L'$.

4. IMPLEMENTATION

In Section 3, Definition 3.3 presents an algorithm to evolve a STTG. The direct implementation of this algorithm leads to an exponential time and space behavior. While that definition eases the presentation, its implementation needs to be efficient, in order to be useful in practice. In this section, we describe a new version of the same algorithm, which is suited for direct implementation. It is straightforward to see that this algorithm generates the same result as that of Definition 3.3. The next algorithm keeps a set of unprocessed non-terminals which are accessible from the start symbol (this set is represented by U in Definition 4.1). In this way we just compute those non-terminal symbols which are accessible from the start symbols of the grammar.

Definition 4.1 (RTG into STTG Transformation) Let $G_0 = (N_0, T_0, S_0, P_0)$ be (general) regular tree grammar. We define a new single-type tree grammar $G = (N, T, S, P)$, obtained from G_0, according to the following steps:

1. Let $S := \{\hat{A}^{S_0} \mid A \in S_0\}$;
2. Let $G := (N := \emptyset, T := T_0, S, P := \emptyset)$;
3. Let $U := S$;
4. While $U \neq \emptyset$ do:

(a) Choose $\{A_1, \ldots, A_n\} \in U$;

(b) Let $U := U - \{\{A_1, \ldots, A_n\}\}$;

(c) Let $N := N \cup \{\{A_1, \ldots, A_n\}\}$;

(d) Let $P := P \cup \{\{A_1, \ldots, A_n\} \rightarrow a\,\sigma_{N(R)}(R) \mid A_1 \rightarrow a[R_1], \ldots, A_n \rightarrow a[R_n] \in P_0, R = (R_1|\cdots|R_n)\}$;

(e) Let $U := (U \cup \{\hat{A}^{N(R)} | A \in N(R)\}) - N$;

End While

5. Return G.

□

In Definition 4.1, we start by computing the equivalence classes of the start symbols of G_0 and we insert them to the set U, containing those non-terminals which are not yet processed. At each iteration of the while loop, an element of U is chosen as the new non-terminal for which a new production rule is going to be created. This non-terminal is added to the set N. At each step the set U is updated by adding to it those non-terminals appearing on the right-hand side of the new production rule, filtering the non-terminals already processed.

Our algorithms are designed to be used in a scenario where a (XML) data administrator needs to change the type of their data. The schema evolution is triggered by the arrival of new documents that are *not* validated by the existing schema. In this case, the administrator will build a simple schema from the new data and add the new rules to those of the existing type. The tree grammar resulting from this operation will not be a valid schema, but it can be seen as an instance of a general regular tree grammar. This grammar will be the input to our algorithms.

We have a prototype tool that implements the proposed algorithms. The purpose of our prototype is to serve as a proof of concept. It was implemented using the ASF+SDF Meta-Environment [vdBHdJ+01] and it is about 1000 lines of code. The grammars presented in the Examples 1.1 and 3.3 were obtained by our prototype. Despite its purpose as a proof of concept, our implementation runs quite fast, since schemas are generally not very large.

5. RELATED WORK

As discussed in [Flo05], traditional tools require the data schema to be developed prior to the creation of the data together with an agreement of the involved communities over a the structure and a vocabulary. Unfortunately, in several modern applications the schema often changes as the information grows and different people, organisations and communities have inherently different ways of modeling the same information. Complete elimination the schema does not seem to be a solution since it assigns meaning to the data and thus helps automatic data search, comparison and processing. To find a balance, [Flo05] considers that we need to find how to automatically map schemas and vocabulary to each other and how to rewrite code written for a certain schema into code written for another schema describing the same domain.

Most existing work in the area of XML schema evolution is placed in the second proposed solution. For instance, in [GMR05] a set of schema update primitives is proposed and the impact of schema updates over XML documents is analysed. The basic idea is to keep track of the updates made to the schema and to identify the portions of the schema that require validation. The document portions affected by those updates are then re-validated (and changed, if necessary).

Our approach aims to be included in the first proposed solution of [Flo05] since it allows the conservative evolution of schemas. Indeed, our method extends the work in [BDH+04, dHM07, BDFM09] which considers the conservative evolution of LTG in a local perspective, *i.e.* by proposing the evolution of regular expressions. Contrary to this, the present paper proposes schema evolution in a global perspective, dealing with the tree grammars as a whole. Moreover, we also consider the evolution of STTG.

Our proposal is inspired in some grammar inference methods (see [Ang92, Sak97] for surveys) that returns a tree grammar or a tree automaton from a set of positive examples. For instance, in [BM03] we find a method for learning a subclass of (ranked) regular tree languages. It is based on the construction of a tree automaton whose states are equivalence classes. In [BM06], the same authors establish that the whole set of (ranked) RTL is efficiently identifiable from membership queries and positive examples. Our method deals with unranked trees, starts from a given RTG G_0 (representing a set of positive examples) and finds the least LTG or STTG that contains $L(G_0)$. As we consider a initial tree grammar we are not exactly inserted in the learning domain, but their methods inspire us and give us tools to solve our problem, namely, the evolution of a original schema (and not the extraction of a new schema).

Several papers (such as in [GGR+00, Chi01, BNST06, BNV07]) deal with XML schema inference. An algorithm for extracting intuitive DTDs from XML document is proposed in [GGR+00]. In [Chi01] XML schemas are modelled as extended context free grammars and the author proposes an extraction algorithm, based on grammar inference, which merges non terminals with similar context. The initial grammar comes from a set of non terminals and production rules as contents of document nodes. In [BNST06] DTD inference consists in an inference of regular expressions from positive examples. As the seminal result from Gold [Gol67] shows that the class of all regular expressions cannot be learnt from positive examples, [BNST06] identifies classes of regular expressions that can be efficiently learnt. Their basic method consists in inferring a single occurrence automaton called SOA from a finite set of strings and to transform it to a SORE (regular expressions in which every element name can occur at most once). Their method is extended to deal with XMLSchema (XSD) in [BNV07]. They introduce the concept of *single occurrence XSD*, noted by SOXSD, as an XSD containing only SOREs. In this context, they propose a method to infer k-local (a XSD having its content models depending only on labels up to k ancestor) SOXDs.

6. CONCLUSION

This paper deals with type evolution of semi-structured data. Our algorithms give schemas generating the *minimal* tree languages containing the original type. Our goal is to allow a given schema to evolve encompassing the needs of the application using it. Therefore, we always consider an initial existing type that should evolve and not the construction of

a schema from scratch.

The paper includes all proofs concerning the correction and the minimality of the generated grammars. A prototype has been implemented in order to show the feasibility of our approach. This work complements the proposals in [BDFM09, dHM07], since we consider not only DTD but also XSD schema, and adopts a global approach where all the tree grammar is taken into account as a whole.

Some aspects of our tools can be improved, in particular the conciseness of the regular expressions appearing in the generated grammars. We believe that techniques used in the DTD extraction domain, such as [GGR+00], can be used in this context. We are considering the comparison of our generated LTGs (or DTDs) with hose obtained by schema extractors.

7. REFERENCES

[Ang92] Dana Angluin. Computational learning theory: survey and selected bibliography. In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 351–369, New York, NY, USA, 1992. ACM.

[BDFM09] Béatrice Bouchou, Denio Duarte, Mirian Halfeld Ferrari, and Martin A. Musicante. Extending xml types using updates. In Dr. Hung, editor, *Services and Business Computing Solutions with XML: Applications for Quality Management and Best Processes*, pages 1–21. IGI Global, 2009.

[BDH+04] B. Bouchou, D. Duarte, M. Halfeld Ferrari, D. Laurent, and M. A. Musicante. Schema evolution for XML: A consistency-preserving approach. In *Mathematical Foundations of Computer Science (MFCS)*, number 3153 in Lecture Notes in Computer Science, pages 876–888. Springer-Verlag, August 2004.

[BM03] Jérôme Besombes and Jean-Yves Marion. Apprentissage des langages réguliers d'arbres et applications. *Traitement automatique de langues*, 44(1):121–153, Jul 2003.

[BM06] Jérôme Besombes and Jean-Yves Marion. Learning tree languages from positive examples and membership queries. *Theoretical Computer Science*, 2006.

[BMW01] A. Brüggeman-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over non-ranked alphabets. Technical Report HKUST TCSC 2001 05, Hong Kong Univ. of Science and Technology Computer Science Center (available at `http://www.cs.ust.hk/tcsc/RR/2001Â-05.ps.gz`), 2001.

[BNST06] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of concise DTDs from XML data. In *VLDB*, pages 115–126, 2006.

[BNV07] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring xml schema definitions from xml data. In *VLDB*, pages 998–1009, 2007.

[CDG+02] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. Available on: *http://www.grappa.univ-lille3.fr/tata*, 1997 (new version 2002).

[Chi01] B. Chidloviskii. Schema extraction from XMLS data: A grammatical inference approach. 2001.

[dHM07] Robson da Luz, Mírian Halfeld Ferrari, and Martin A. Musicante. Regular expression transformations to extend regular languages (with application to a datalog XML schema validator). *Journal of Algorithms (Special Issue)*, 62(3-4):148–167, 2007.

[Flo05] Daniela Florescu. Managing semi-structured data. *ACM Queue*, 3(8):18–24, 2005.

[GGR+00] Minos N. Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. Xtract: A system for extracting document type descriptors from xml documents. In *SIGMOD Conference*, pages 165–176, 2000.

[GMR05] Giovanna Guerrini, Marco Mesiti, and Daniele Rossi. Impact of XML schema evolution on valid documents. In *WIDM'05: Proceedings of the 7th annual ACM international workshop on Web information and data management*, pages 39–44, New York, NY, USA, 2005. ACM Press.

[Gol67] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[KH08] John Kavanagh and Wendy Hall, editors. *Grand Challenges in Computing Research Conference*. UK Computing Research Committee, 2008. `http://www.ukcrc.org.uk/grand_challenges/news/gccr08final.pdf`.

[ML02] Murali Mani and Dongwon Lee. Xml to relational conversion using theory of regular. In *In VLDB Workshop on EEXTT*, pages 81–103. Springer, 2002.

[MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Inter. Tech.*, 5(4):660–704, 2005.

[Sak97] Yasubumi Sakakibara. Recent advances of grammatical inference. *Theor. Comput. Sci.*, 185(1):15–45, 1997.

[vdBHdJ+01] Mark van den Brand, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter Olivier, Jeroen Scheerder, Jurgen Vinju, Eelco Visser, and Joost Visser. The asf+sdf meta-environment: a component-based language development environment. *Electronic Notes in Theoretical Computer Science*, 44(2), 2001.